RandPeer: Membership Management for QoS Sensitive Peer-to-Peer Applications

Jin Liang and Klara Nahrstedt
Department of Computer Science
University of Illinois at Urbana-Champaign
{jinliang, klara}@cs.uiuc.edu

Abstract

Many Peer-to-peer (P2P) applications such as media broadcasting and content distribution require a high performance overlay structure in order to deliver satisfying quality of service (QoS). Previous approaches to building such overlays either involve a shared contact point, which results in non-scalable solutions, or rely on gossip style membership dissemination, which lacks QoS awareness. In this paper, we present a distributed membership service called RandPeer*, which manages membership information on behalf of P2P applications, and allows peers to locate good neighbors based on their QoS characteristics. Using this service, P2P applications can easily construct their overlays in a scalable and QoS aware fashion.*

We have implemented RandPeer and experimented in both local and wide area environments. Our results show that (1) RandPeer is scalable and robust to highly dynamic P2P memberships; (2) RandPeer has good lookup performance, both in terms of response time and the randomness of peer selection. The latter improves load balancing and failure resilience of P2P applications; (3) when used to improve the performance of a mesh based P2P overlay, RandPeer achieves 10% improvement in just 2 protocol rounds, which is more than 5 times faster than pure random neighbor selections.

1 Introduction

Unlike early file sharing applications such as Napster and Gnutella, many recent P2P applications, including live media broadcasting [1, 2, 3, 4, 5], high bandwidth content distribution [6, 7], and real-time audio conferencing [8] require a high performance overlay structure in order to deliver satisfying Quality of Service (QoS) to the users. As a result, building high performance overlays is an important task for these applications.

Ideally, if the characteristics of all the application peers are known, a globally optimal structure can be built using some deterministic algorithms [1, 9, 2]. This holistic approach might produce the best possible overlay structure. However, it is not scalable. This is because P2P applications often consist of large number of peers, which may join and leave at any time. Maintaining consistent state for such dynamic systems would incur too much overhead. Therefore, most recent systems have adopted a localized approach, where each peer joins the overlay by independently selecting its neighbors, and neighbor failures are repaired by local adjustment. Clearly, the performance of an overlay built this way depends critically on the ability of individual peers to select good neighbors. Thus, the question becomes how can individual peers select their neighbors in a scalable and QoS aware fashion?

In some systems such as NICE [3] and Zigzag [4], membership information is embedded in the overlay structure. To select its neighbors, a joining peer first contacts a well known rendezvous point, then successively probes existing peers until it finds its best position in the overlay. While this approach allows peers to locate good neighbors, it has two drawbacks. First, for highly dynamic systems, the contact point can easily become a bottleneck. Second, when the system is large, a joining node may have to probe many peers before finding its best position, which is also undesirable.

To address the scalability problem, many recent systems such as DONet [5], PRO [10] and later versions of Narada [11] employ a gossip style membership management scheme similar to [12]. The idea is that each peer maintains a small list (called the partial view) of other members in the system, and periodically exchanges membership information with other peers. As a result, the partial view represents a uniform sampling of the whole system. Whenever a neighbor is needed, it is selected randomly from the partial view.

Although the gossip scheme achieves scalability in

membership management, it is not suited for QoS aware neighbor selection. This is because a randomly selected peer in a large system is unlikely to have the desired QoS characteristics (e.g., delay and bandwidth). The gossip style membership management was originally designed for probabilistic data dissemination applications [12], which are themselves gossip based applications. It is not clear how the scheme can be modified to accommodate different application QoS requirements. In addition, because a peer can learn about other peers indirectly (through gossip), it is difficult to detect if some peers in the partial view have failed.

In this paper, in recognizing that QoS aware neighbor selection, and hence membership management is common to QoS sensitive P2P applications, we propose to *decouple membership management from P2P applications* by designing a distributed membership service called *RandPeer*. RandPeer allows decentralized membership registration for peers, and enables efficient lookup of other peers. In its basic form, RandPeer allows the lookup of a peer that is randomly selected from the entire system. Randomness in neighbor selection is necessary in a P2P environment, because it avoids the overhead for deterministic search of the best neighbor. However, different from the gossip based scheme [12], RandPeer can also cluster peers based on their QoS characteristics. By restricting random peer selection within specific QoS clusters, RandPeer achieves QoS awareness in neighbor selection, while at the same time preserves the efficiency offered by randomness.

We have implemented the RandPeer service and conducted experiments in both local and wide area (PlanetLab [13]) environments. Our results show that (1) RandPeer is scalable and robust to both bulk and constant membership changes; (2) RandPeer has good lookup performance, both in terms of response time and randomness in peer selection. The former is bounded by $O(\log \log N)$, where N is the maximum number of peers in the system, and the latter is close to uniform distribution, which improves the load balancing and failure resilience of P2P applications; (3) when used to improve the performance of a mesh based P2P overlay, RandPeer achieves 10% improvement in just 2 protocol rounds, which is more than 5 times faster than pure random neighbor selections. This demonstrates the importance of QoS awareness in P2P neighbor selections.

In the rest of this paper, we first describe the application model that RandPeer targets in Section 2, then present the detailed design of RandPeer in Section 3. Section 4 is the evaluation results. Section 5 discusses related research and Section 6 concludes the paper.

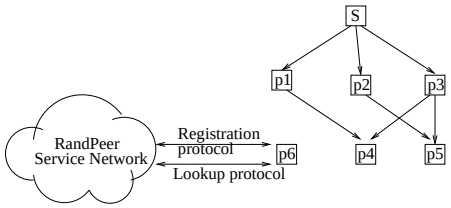


Figure 1. App. scenario that uses RandPeer

2 Application Model

In this paper, we consider QoS sensitive P2P applications such as live media streaming and high bandwidth content distribution, which can be primarily characterized by their large and dynamic memberships, and their requirement on high performance overlay structures. We assume the localized overlay construction and maintenance approach is used. This means both the initial joining of a peer and the recovery from a neighbor failure are done by individual peers selecting their neighbors. RandPeer provides support to such applications by managing their membership in a scalable and robust way, and allowing peers to select neighbors based on their QoS requirements.

Figure 1 shows how an example live P2P streaming application can benefit from RandPeer. A single source S in the application provides the live content (e.g., Internet TV program). Other peers (receivers) can join the system and stream the content from either the source or other peers. In line with recent work on P2P streaming [14, 10, 5], we allow each peer to stream from multiple parent peers in order to better utilize their residue bandwidth.

When a new peer (e.g., $p6$ in Figure 1) joins the system, it needs to locate some other peers suitable as its parents. Without a membership service, the source S would necessarily become a shared contact point. With RandPeer, however, each peer currently in the system (including the source) can register with the service using a registration protocol, and a joining peer ($p6$) can query the service [1] to lookup potential good parents, and connect to these parents. At the same time, the joining peer can register with the RandPeer service, so that it can be located by other peers. Later, if a parent fails or experiences degraded performance, a similar lookup can be made to locate an alternative parent.

Thus we can see by decoupling membership management from P2P applications, RandPeer greatly facilitates their development. The applications only need to

[1] RandPeer is a distributed service that consists of different service nodes. We assume each peer knows about a service node via some out of band information, in a way similar to using the domain name service (DNS).

Table 1. System parameters

h	max number of bits in a bin label
b	number of bits in a peer ID/lookup key
B	capacity of a leaf bin
N	max number of peers in an application
m	random ratio

invoke some stub code provided by RandPeer (which executes the registration and lookup protocols), without knowing how the registration information is organized, and how the lookup is performed. Since RandPeer is offered as a service, it has the additional benefit that a single service can be shared by multiple applications, and that the service can evolve (e.g., be reimplemented for better performance) without changing the applications, so long as the protocols for accessing the service remain unchanged.

The challenge, however, is to design the service so that it manages P2P membership information in a scalable and robust manner, and allows efficient, QoS aware neighbor selections. To meet the challenge, RandPeer uses a distributed trie data structure to organize membership information for P2P applications, and employs simple but robust protocols to dynamically grow and shink the trie as peers join and leave [2]. Finally, RandPeer allows peers to translate their QoS characteristics to QoS prefixes in their ids. This enables RandPeer to cluster peers based on their QoS prefixes. QoS aware neighbor selection can then be achieved by restricting peer lookups to specific QoS clusters.

3 Design of RandPeer Service

3.1 Membership Trie

RandPeer uses a trie data structure to organize membership information for P2P applications. A trie is basically a tree with its nodes labeled with 0, 1 strings [3]. The label of the root is an empty string. If a node has label l, its left child is labeled $l0$, and its right child is labeled $l1$. Each node in the trie is called a "bin". Peer registration information is only stored at leaf bins. We assume the maximum length of a label is h, thus there are at most 2^h leaf bins in a membership trie (a list of the system parameters is given in Table 1). Figure 2 shows an example membership trie. Note the membership trie is a data structure within

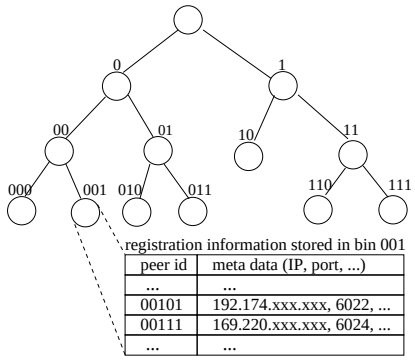


Figure 2. Membership trie

the RandPeer service to organize the registration information for application peers. It's different from the actual overlay network formed by the peers.

To register its membership with the trie, a peer must randomly select a *peer id* of b bits ($b \geq h$) for itself. This peer id determines which leaf bin the peer should register with. Specifically, it should register with the leaf bin whose label is a prefix of its peer id. Clearly, given a peer id, there exists a unique leaf bin whose bin label is a prefix of the peer id. As an example, in Figure 2, peers 00101 and 00111 should register with bin 001, because this is the leaf bin whose label is a prefix of their peer ids. Similarly, a peer with id 10010 should register with leaf bin 10.

The membership trie is a dynamic data structure. Each leaf bin has a capacity B, which means it can store at most B registration entries. If there are more than B peers registered with a leaf bin (with a label l), the bin can be *split* into two leaf bins (with labels $l0$ and $l1$, respectively). All peers previously registered with bin l are now informed to register with its children bins. If later the total number of peers registered with $l0$ and $l1$ drops to below $B/2$, the two leaf bins can be *merged* again. The split of bins ensures that no leaf bin is overloaded by too many registrations, and the merge of bins improves the lookup performance by removing sparsely populated leaf bins.

RandPeer is built on top of a distributed hash table (DHT), therefore we use consistent hashing to map the logic membership trie to the RandPeer service nodes. Figure 3 shows how the mapping is done. The bottom of the figure shows the RandPeer service network, which consists of distributed RandPeer service nodes [4]. Given a bin in the membership trie (e.g., bin 1001 in Figure 3), we use a consistent hashing function such as SHA1 [16] to map it to a unique number in the DHT

[2]The dynamic nature of the membership information makes the service significantly different from other services such as resource discovery, pub/sub, etc.

[3]For simplicity, we only discuss binary tries.

[4]The RandPeer nodes self-organize into an overlay, which is a ring if Chord [15] is used as the DHT.

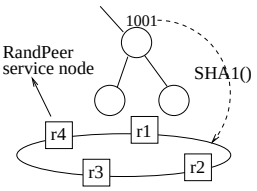


Figure 3. Mapping from membership trie to RandPeer nodes

key space. This number is called the *bin id* of the bin. The "successor" [15] of this bin id (which is node $r2$ in this case) is responsible for maintaining the state in the bin. Since each application has its membership trie, to avoid conflict between the membership bins of different applications, we assume each application has a unique *application id*, and include it in the hashing process. To send a message to bin 1001, a peer can obtain its bin id using the same mapping process, and send the message to a nearby RandPeer node. The message is then routed (using the underlying DHT routing mechanism) to the RandPeer node (e.g., $r2$), which processes the message.

3.2 Registration Protocol

Due to the dynamic nature of P2P memberships, RandPeer takes a soft state approach to membership management. Specifically, the registration information for each peer has a life time. It must be refreshed periodically (by `Register` messages), otherwise it will time out and be removed from its leaf bin.

When a new peer begins to register, it must first locate its leaf bin. This can be done by walking down the trie, starting from the root bin, until a leaf bin is reached. However, this would suffer from the shared contact problem of NICE [3] and Zigzag [4]. Therefore, RandPeer uses the binary search algorithm described in Section 3.3 to quickly locate the leaf bin for a peer in $O(\log \log N)$ steps. Once the leaf bin is located, the peer begins to periodically send `Register` messages to its leaf bin. The leaf bin is expected to send a `RegisterOK` message back, unless it has been marked for split or merge (to be discussed below). In this case a `RegisterGoDown` or `RegisterGoUp` message is sent back, and the peer will move to its new leaf bin.

Each `Register` message contains the peer id of a peer and some meta information such as its IP address and port number. This information is stored in the leaf bin as illustrated in Figure 2. Each `RegisterOK` message also contains the meta information previously registered under the peer id. This can be used to detect

```
label_bits = TryRegister(peer_id) //locate initial leaf bin
bin_label = peer_id ≫ (b − label_bits)
bin_id = SHA1(app_id, bin_label)
while(true)
  send_message(Register, bin_id, meta_info)
  msg = receive_message() //reply message from leaf bin
  if msg.type == RegisterOK
    if msg.is_leader == true
      if msg.num_entries ≥ B SendSplitMsg()
      else if msg.num_entries < B/4 SendMergeMsg()
    end if
  else
    if msg.type == RegisterGoDown label_bits++
    else label_bits−−
    bin_label = peer_id ≫ (b − label_bits)
    bin_id = SHA1(app_id, bin_label)
  end if
  sleep //wait till next protocol period
end while
```

Figure 4. Registration Protocol

if two peers have chosen the same peer id.

The information stored at each interior bin (e.g., the height of the subtree rooted at the interior bin) is also soft state. Therefore, each leaf bin and interior bin (except the root) also needs to periodically refresh its parent bin using a `Refresh` message.

Bin Split/Merge. To dynamically grow and shrink the membership trie, our registration protocol relies on the coordination between RandPeer and the application peers for bin split/merge. Each leaf bin has a *leader peer*, which is the peer with the smallest peer id in the bin. The leader peer of a leaf bin is responsible for initiating the bin split/merge process. Each `RegisterOK` message contains a bit indicating if the peer is the leader peer. It also contains the number of registration entries in the bin. If the leader peer finds the number of entries $\geq B$, it will send a `Split` message to the bin. If the bin accepts the message, it will mark itself for split. Any peer that attempts to register with such a bin will receive a `RegisterGoDown` message, which causes the peer to register at a lower level and new leaf bins to be created. The "marked for split" state is a temporary state and the bin will become an interior bin after a short time.

If a leader peer finds the number of entries in a leaf bin to be too small (e.g., $< B/4$), it will send a `Merge` message to the parent bin. If the `Merge` message is accepted, the parent bin is marked for merge. If a bin marked for merge receives a `Refresh` message from its child bin, it will reply with a `Terminate` message. The child will then respond to any `Register` message with

a `RegisterGoUp` message, telling the peers to register with a higher level. The "marked for merge" state is also temporary and the bin will become a leaf bin after a short time.

Both the `Split` and `Merge` message may or may not successfully mark the bin. For example, the message may be lost, or for `Merge` messages, the other child of the parent bin may not be a leaf bin. For example, in Figure 2 if the leader peer of bin 10 sends a `Merge` message to bin 1, it will be ignored. This is because bin 11 is not a leaf, which means there may be more than B peers registered under the subtree rooted at bin 1. As a result, after a leader peer has sent a `Split` or `Merge` message, it continues with its normal registration process. If the split/merge is successful, the leader peer will be notified to go up or down, just like other peers. If it is not successful, the leader peer will retry periodically, at a low frequency, as long as the condition for split/merge remains. This approach greatly simplifies the protocol between the RandPeer service and the application peers. It also makes the protocol robust, because the eventual success of bin split/merge is not affected by the loss of one or more messages.

The registration protocol (as executed by the registration stub at each peer) is shown in Figure 4. In Figure 4, *TryRegister* uses the binary search algorithm in Section 3.3 to quickly locate the initial leaf bin. b and B are the system parameters as given in Table 1.

3.3 Random Peer Lookup

In this subsection, we present the algorithm to lookup a random peer that is currently registered with RandPeer. Random peer lookup avoids the overhead of deterministic search, and improves load balancing and failure resilience of P2P applications. In the next subsection, we will describe how RandPeer clusters peers for QoS aware neighbor selections.

To look up a random peer, a peer generates a random *lookup key* of b bits, and sends a `Lookup` message to the leaf bin whose label is a prefix of the lookup key. The leaf bin will return the registration information of the registered peer whose id immediately follows the lookup key. For example, in Figure 2, if the lookup key is 00100, the `Lookup` message will be sent to bin 001. Since the peer id 00101 is the one that immediately follows the lookup key, the registration information for this peer will be returned.

To quickly locate the leaf bin, we use an algorithm similar to binary search. Initially, the query peer sends a `Lookup` message to a bin whose bin label is a prefix of the lookup key, and has a length (*label_bits*) of $h/2$. If the bin is a leaf bin, a `LookupOK` message is returned, together with the lookup result. Otherwise a `LookupGoDown` or `LookupGoUp` message is returned, depending on if the bin is an interior bin, or does not exist. The query peer will then change the *label_bits* (with exponentially decreasing step size), and retry other bins. The lookup protocol as executed by the query peer is shown in Figure 5.

```
label_bits = h/2
step_size = label_bits/2
bin_label = lookup_key ≫ (b − label_bits)
bin_id = SHA1(app_id, bin_label)
while(true)
  send_message(Lookup, bin_id, lookup_key)
  msg = receive_message()
  if msg.type == LookupOK
    return msg.result
  if msg.type == LookupGoUp label_bits −= step_size
  else label_bits += step_size
  bin_label = lookup_key ≫ (b − label_bits)
  bin_id = SHA1(app_id, bin_label)
  if step_size > 1 step_size /= 2
end while
```

Figure 5. Lookup Protocol

It is possible that the lookup key is larger than any peer id stored in the leaf bin. In this case, no peer is returned in `LookupOK` and the querying peer should try the "next" leaf bin. What it does is to increase the lookup key, so that it just falls out of the current leaf bin, and repeat the lookup process. As an example, in Figure 2, if a peer generates a random lookup key 01111, the `Lookup` message will be sent to leaf bin 011. If every peer registered in the bin has an id smaller than 01111, the query peer will increase the lookup key to 10000, which just falls out of the bin 011. This time the `Lookup` message will be sent to leaf bin 10, and the first peer in this bin will be returned.

Clearly the protocol in Figure 5 can locate a leaf bin in $O(\log h)$ steps. Since $h = O(\log N)$, where N is the maximum number of peers in the system, it means the protocol can locate the leaf bin in $O(\log \log N)$ steps, regardless the number of peers currently in the system. To further improve the lookup performance, we can let each interior bin record the minimum height of its left and right subtrees. When a `Lookup` message is received by an interior bin, it will return the minimum height of its subtree together with the `LookupGoDown` message. The query peer can use this information to reach the leaf bin more quickly.

In the above registration/lookup protocol, both the peer registration id and lookup key are randomly selected, and a peer is returned if its id immediately fol-

lows a lookup key. It is well known such an approach may not result in uniformly random peer selection. In fact, with high probability, some peers may be selected t times more often than other peers [15], where t is logarithmical in the current system size. This is because even though each peer selects an id randomly, the resulting membership trie may not be perfectly balanced. There are many load balancing techniques that can address the problem [17, 18]. For simplicity, we use a simple heuristic to improve the randomness of the lookup result. Each time a random peer is needed, we look up the m peers whose ids immediately follow the lookup key, we then randomly choose one from the m peers to be the final random peer. m is called the *random ratio* of the lookup. Since each leaf bin visited may contain multiple entries following the lookup key, looking up m peers does not significantly increase the lookup overhead.

3.4 QoS Aware Neighbor Selection

The basic membership registration and lookup protocols described above allow the lookup of random peers. To build high performance overlays, however, we may want to look up peers based on some given QoS metrics. This can be achieved by combining application specific QoS metrics with peer id and lookup key selection. Specifically, the id of a peer is divided into a *QoS prefix* and a random suffix. The QoS prefix encodes its QoS characteristics, and determines the possible set of leaf bins that the peer can register with. If two peers have the same QoS characteristics, they will be automatically clustered under the same subtree in the membership trie. When a peer needs to look up some other peer, it generates a random key with the desired QoS prefix. The lookup result of such a key is likely to be a peer with the desired QoS characteristics.

As an example, suppose a P2P application wants to minimize the average delay between neighboring peers, we can use a QoS prefix (e.g., 5 bit binary string) for each peer that indicates its geographical location [5]. When a peer needs to look up a neighbor, it generates a lookup key that has the same prefix as its own peer id. The result of such a lookup is likely to be a peer that is close by. As another example, suppose a P2P application wants to select neighbors based on their access bandwidth, we can use a prefix to encode the access bandwidth of the peers. To look up a peer that has certain access bandwidth, we can generate a lookup key with the specific prefix. The prefix can also encode multiple QoS metrics, so that we can lookup a peer that is not only close by, but also has the desired access bandwidth.

[5]Such prefixes can be generated, for example, using the landmark binning technique introduced in [19].

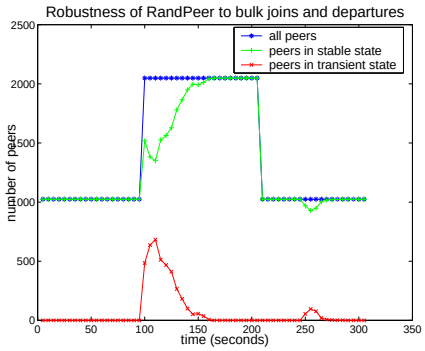


Figure 6. Robustness of RandPeer to bulk joins and departures

We note that RandPeer provides the ability to cluster application peers based on their QoS prefixes, and to lookup neighbors from a specific cluster of peers. However, it is up to the application to decide how its QoS metrics should be translated into QoS prefixes. Different applications may have different QoS requirements and different translation schemes. All these are transparent to the registration and lookup protocols.

4 Performance Evaluation

We have implemented RandPeer on top of the Chord [15] code that we obtained from the I3 project [20]. Since RandPeer can be implemented using any DHT, in our evaluation, we focus on the performance of RandPeer itself, instead of that of the underlying DHT. Specifically, we want to examine (1) the scalability and robustness of RandPeer to highly dynamic P2P memberships; (2) the performance of random peer lookup, both in terms of response time and the randomness of the lookup result; (3) the impact of QoS aware neighbor selection on the performance of P2P overlays. Most of our experiments are run in a local environment, which means the RandPeer service is started on a single machine. For the lookup performance experiments, we also deploy RandPeer on about 20 PlanetLab [13] nodes. For all experiments, we set $h = 16$, $b = 32$ and $B = 16$.

4.1 Robustness of RandPeer

Figure 6 shows the performance of RandPeer for bulk peer joins and departures. A peer is said to be in stable state, if its leaf bin is not under split or merge

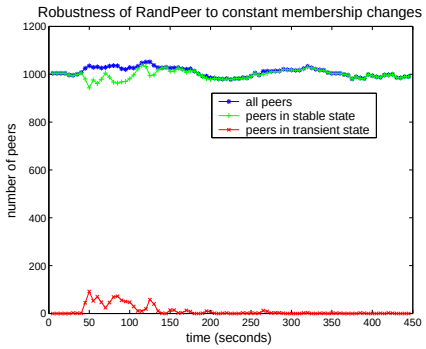


Figure 7. Robustness of RandPeer to constant joins and departures (churns)

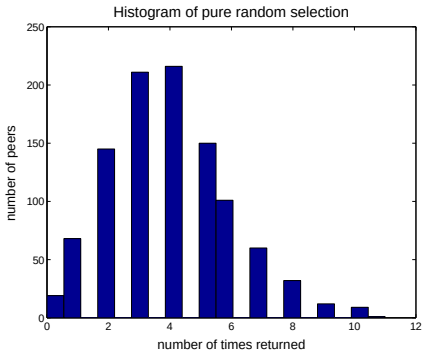


Figure 9. Histogram of pure random selection.

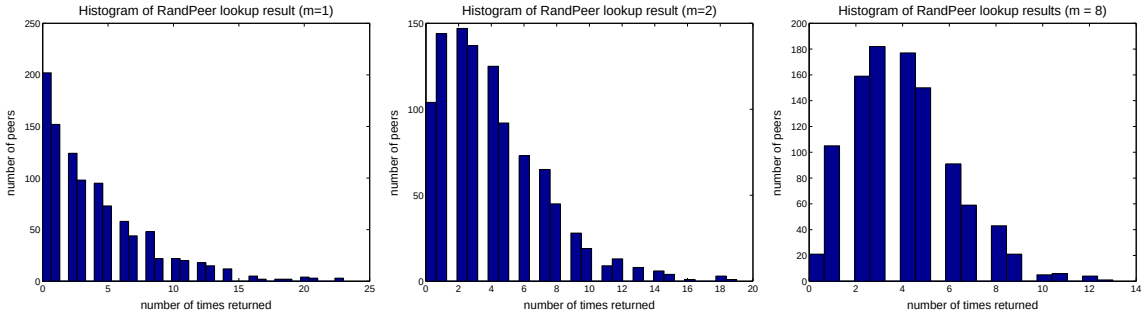
process. Otherwise it is said to be in transient state. We first register 1024 peers with the RandPeer service. After the system is stabilized, we register another 1024 peers simultaneously at time 95. This causes many leaf bins to be split, and the corresponding peers to register with new leaf bins. However, after only about 70 seconds, all the peers successfully settled down with their new leaf bins. Note in our implementation, each peer refreshes its membership about every 10 seconds. This means for a bulk join of 1024 peers, our system stabilizes in only about 7 protocol periods. This is due to the quick join process described in Section 3.2. At time 205, all the new peers are killed at the same time. Figure 6 shows that after about 40 seconds, their registration entries are timed out. This causes some leaf bins to be merged and the peers to register higher in the membership trie. However, it takes only about 30 seconds for these peers to reach stable state again. The 40 second delay is caused by two facts. First, the registration timeout value is set to a little more than 20 seconds, in order to tolerate occasional message losses. Second, to prevent frequent bin split/merge process, a leader peer will send a `Split` or `Merge` message after the condition for split/merge has been true for two protocol periods. Although this causes some delay in bin split/merge, it improves the stability of RandPeer.

Figure 7 shows the robustness of RandPeer to continuous peer joins and departures (churns). We register 2048 peers with RandPeer. Initially half of the peers are registered and in stable state. Starting from time 20, all peers begin to switch between on and off states. Both on and off periods are exponentially distributed with a mean of 300 seconds. When off peers come back, they will register with new peer ids. Figure 7 shows that initially some peers are affected by the joining and departure of other peers. However, after time 150, most nodes are settled in stable state, even though during each second there are about 7 node join and departure events. This is because each leaf bin can accommodate a range of peer registrations, thus the joining and leaving of a peer does not necessarily cause a bin split/merge. The delay of bin split/merge described above also avoids unnecessary splits/merges.

4.2 RandPeer Lookup Performance

For P2P applications, it is likely that many peers are potential good neighbors for a given peer (for example, they all have the same prefix in their ids). In this case, we should return each peer with equal probability, in order to improve load balancing and failure resilience of the applications. Figure 8 shows the randomness of RandPeer lookup results. For this experiment, we register 1024 peers with the RandPeer service, and then perform 4096 random lookups. Figure 8(a) shows that when the random ratio m is 1, the returned peers are very unevenly distributed. For example, about 200 peers are never selected, while some peers are selected more than 20 times. Figure 8(b) shows that when m is 2, the randomness of the returned peers is still not uniformly distributed. However, Figure 8(c) shows that when $m = 8$, the randomness of the lookup results is much better. Only about 20 peers are not selected, and most peers are selected between 1 and 7 times. In fact, Figure 9 shows the histogram of perfect (uniformly distributed) random selections. We can see Figure 8(c) is very close to pure random selections.

Another aspect of the lookup performance is the response time. To examine the response time of RandPeer lookups, we deploy RandPeer on about 20 PlanetLab nodes (mostly located in North America). Each time we register a given number of peers with RandPeer, and perform 200 random peer lookups, using different random ratio m. We record the average number of DHT lookups (each `Lookup` message incurs one



(a) m = 1 (b) m = 2 (c) m = 8

Figure 8. Randomness of RandPeer lookup results

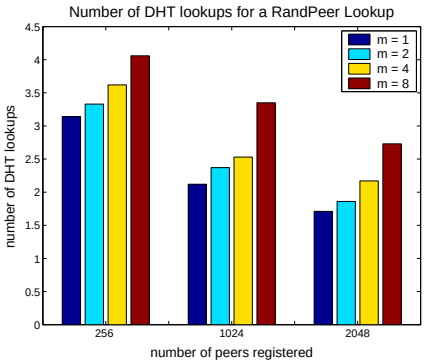


Figure 10. Number of DHT lookups for a RandPeer lookup

DHT lookup) that is needed, and the actual delay (in milliseconds) to get the result. The number of DHT lookups reflects RandPeer's performance independent of the underlying DHT. The delay reflects the performance of our particular implementation.

Figure 10 shows the number of DHT lookups needed for a RandPeer lookup. First we can see that larger random ratio involves more DHT lookups. This is because we need to locate m peers in order to produce a random lookup result. However, even for $m = 8$, only 3 or 4 DHT lookups are needed. Second, the number of DHT lookups is the largest when there are 256 peers in the system, and the smallest when there are 2048 peers. The reason is that when there are 256 peers, the leaf bins have a bin label of 5 to 6 bits. According to the lookup protocol in Figure 5, a query peer will first try a bin with $label\_bits = h/2 = 8$, then $label\_bits = 4$, and then reach the leaf bin. So on average, it takes a little more than 3 DHT lookups (for $m = 1$). When the system has 1024 peers, the leaf bins have 7 to 8 bits. Therefore, the query either succeeds on the first message, or takes 2 or 3 messages (using the height information). When there are 2048 peers, the leaf bins have 8 to 10 bits. Therefore, most lookups take only 1 or 2 messages. These results show that judicious selection of the initial $label\_bits$ in the lookup protocol (e.g., using cached $label\_bits$ from previous lookups) may have an impact on the lookup performance. However, we do not explore this further in this paper.

The delay results are not shown here for space reasons. However, most of the RandPeer lookups take just 200 to 300 milliseconds. Even for 256 peers and $m = 8$, we can locate a random peer in about 380ms. We believe this is acceptable for a control plane operation. Note the Chord code we used is not locality aware. If we use some other DHT such as Pastry [21], we would expect even better lookup delays.

4.3 Impact of QoS Aware Neighbor Selection

To examine the impact of QoS aware neighbor selection on P2P applications, we simulate a mesh based P2P application, and see how RandPeer can help reduce the average delay between neighboring peers.

For this experiment, we use the BRITE [22] topology generator to generate a two level hierarchical network topology that consists of 10000 nodes. We then randomly select a subset of nodes as peers in the P2P application. We also randomly select several landmark nodes and use the binning technique in [19] to generate landmark vectors as prefixes for the peers. We use two bits to encode the delay of a node to each landmark node. Peers will generate their ids with the given prefixes and register with the RandPeer service.

For each experiment, we first build a pure random

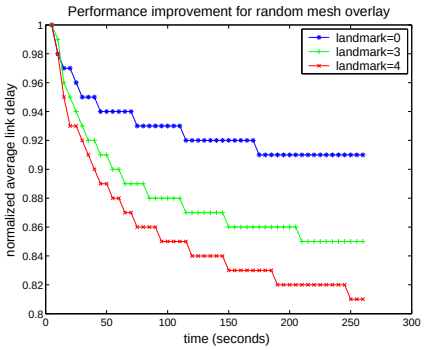


Figure 11. Performance improvement for an application of 1024 peers

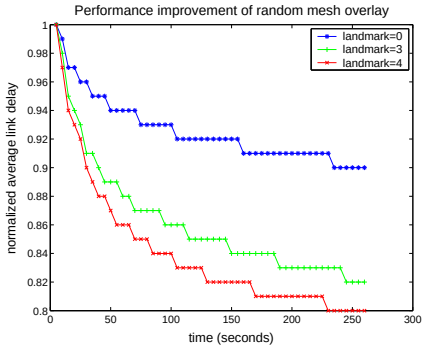


Figure 12. Performance improvement for an application with 4096 peers

mesh of the peers. After that, each peer can periodically (with an average of 20 seconds) select a random peer and replace its worst neighbor with the selected peer, if the selected peer is closer (has a smaller delay) than the worst neighbor. We use two methods for peer selection. The first uses no landmarks, which corresponds to pure random selection. The second uses RandPeer to select a node with the same prefixes. Every 5 seconds, we evaluate the overall average link delay of the mesh, and compare it with that of the initial random mesh.

Figure 11 shows the performance ratio for 1024 peers. We can see that using landmark vectors to cluster the peers can improve the performance of the application much faster than pure random peer selection (landmark=0). For example, it takes about 50 seconds for the application to reduce the average link delay to 90% of the initial mesh, if 3 landmark nodes are used. It takes only 40 seconds (two evolvement rounds) if 4 landmarks are used. However, for pure random peer selection, even after 250 seconds, the delay improvement is still less than 10%. By that time, RandPeer has improved the performance of the application by 15% if 3 landmarks are used, and 19% if 4 landmarks are used.

Figure 12 shows similar results for a larger system of 4096 peers, except the initial performance improvement of RandPeer is even faster. This is because the larger a system is, the less likely that a randomly selected peer would be a good neighbor. By clustering peers based on their characteristics such as geographical location, RandPeer can focus on a much smaller set of peers and return much better peers on average.

5 Related Work

A lot of research on DHTs [15, 21] has attempted to build a generic "routing layer" for large distributed applications. While this has benefited many distributed applications, the multi-hop routing of DHTs is undesirable for QoS sensitive P2P applications such as media streaming. For these applications, we believe a better way is to provide some control plane service to facilitate the applications, while allow the applications to manage their own overlay construction, based on their own QoS requirements.

Membership management is a control plane service that can greatly benefit QoS sensitive P2P applications. Previous approaches either require global membership information [1, 2, 9], or embed membership information in the overlay structure [3, 4], both of which would introduce scalability problems. The gossip style membership management adopted by recent systems [5, 10, 11] can not be easily modified to accommodate different application QoS requirements. In contrast, we have shown that RandPeer achieves both scalability and QoS aware neighbor selections for P2P applications. Some previous work such as CollectCast [14] directly uses DHT to store membership information of some peers (e.g., all the peers that have a particular video file). This scheme lacks the adaptive property of a dynamic trie data structure. As a result, some DHT nodes might be overloaded if many peers have the same video file.

Our use of a trie data structure for membership management bears resemblance to recent research on supporting range query over DHTs such as PHT [23, 24, 25]. However, the highly dynamic nature of membership information means that our protocol must be designed with robustness in mind. For example, we rely on the periodical retry of leader peers and the marking of membership bins to achieve bin split/merge, which results in very simple and robust protocols, yet at the same time avoids the concurrency problem of PHT [24].

6 Conclusion

We have presented the design and evaluation of the RandPeer membership management service for QoS sensitive peer to peer applications. RandPeer achieves scalable and decentralized membership management by using a trie data structure and map the trie to an underlying distributed hash table (DHT), and it enables QoS aware neighbor selection by clustering application peers based on their QoS characteristics. Our experiment results in both local and wide area environments verified that RandPeer is highly robust to dynamic P2P memberships, and has good random peer lookup performance. Further, when used for neighbor selection in a mesh based P2P applications, RandPeer achieves much faster performance improvement than pure random neighbor selections, especially for large systems.

References

[1] Yang hua Chu, Sanjay G. Rao, and Hui Zhang, "A case for end system multicast," in *Proceedings of ACM SIGMETRICS*, June 2000.

[2] Venkata N. Padmanabhan, Helen J. Wang, and Philip A. Chou, "Distributing streaming media content using cooperative networking," in *NOSSDAV'02*, 2002.

[3] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy, "Scalable application layer multicast," in *Proceedings of ACM SIGCOMM'02*, August 2002.

[4] Duc A. Tran, Kien A. Hua, and Tai Do, "Zigzag: An efficient peer-to-peer scheme for media streaming," in *IEEE INFOCOM'03*, 2003.

[5] Xinyan Zhang, Jiangchuan Liu, Bo Li, and Tak-Shing Peter Yum, "DONet: A data-driven overlay network for efficient live media streaming," in *IEEE INFOCOM'05*, Miami, FL, 2005.

[6] Jeannie Albrecht Dejan Kostic, Adolfo Rodriguez and Amin Vahdat, "Bullet: High bandwidth data dissemination using an overlay mesh," in *SOSP'03*, 2003.

[7] John W. Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost, "Informed content delivery across adaptive overlay networks," in *Proceedings of ACM SIGCOMM'02*, August 2002.

[8] Roger Zimmermann and Leslie S. Liu, "Active: Adaptive low-latency peer-to-peer streaming," in *MMCN 05*, 2005.

[9] Minseok Kwon and Sonia Fahmy, "Topology-aware overlay networks for group communication," in *NOSSDAV 2002*, May 2002.

[10] Reza Rejaie and Shad Stafford, "A framework for architecting peer-to-peer receiver-driven overlays," in *NOSSDAV'04*, 2004.

[11] Yang hua Chu, Aditya Ganjam, T. S. Eugene Ng, Sanjay G. Rao, Kunwadee Sripanidkulchai, Jibin Zhan, and Hui Zhang, "Early experience with an internet broadcast system based on overlay multicast," in *Proceedings of USENIX Annual Technical Conference*, Boston, MA, June 2004.

[12] P. Th. Eugster, R. Guerraoui, S. B. Handurukande, A.-M. Kermarrec, and P. Kouznetsov, "Lightweight probabilistic broadcast," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 4, pp. 341–374, November 2003.

[13] "Planetlab," http://www.planet-lab.org/.

[14] Mohamed Hefeeda, Ahsan Habib, Dongyan Xu, Bharat Bhargava, and Boyan Botev, "CollectCast: A peer-to-peer service for media streaming," in *ACM Multimedia'03*, 2003.

[15] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of ACM SIGCOMM'01*, 2001.

[16] "RFC 3174, us secure hash algorithm 1 (sha1)," http://www.ietf.org/rfc/rfc3174.txt.

[17] Gurmeet Singh Manku, "A randomized ID selection algorithm for peer-to-peer networks," in *ACM PODC 2004*, July 2004.

[18] John Byers, Jeffrey Considine, and Michael Mitzenmacher, "Simple load balancing for distributed hash tables," in *IPTPS'03*, February 2003.

[19] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker, "Topologically-aware overlay construction and server selection," in *INFOCOM'02*, 2002.

[20] "Internet indirection infrastructure (i3) web site," http://i3.cs.berkeley.edu.

[21] Antony Rowstron and Peter Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Middleware 2001*, November 2001.

[22] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers, "BRITE: An approach to universal topology generation," in *In Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems-MASCOTS '01*, 2001.

[23] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph Hellerstein, and Scott Shenker, "Brief announcement: Prefix hash tree," in *ACM PODC'04*, July 2004.

[24] Yatin Chawathe, Anthony LaMarca, Sriram Ramabhadhran, Sylvia Ratnasamy, Joseph Hellerstein, and Scott Shenker, "A case study in building layered DHT applications," Tech. Rep. IRS-TR-05-001, Intel Research, January 2005.

[25] Jun Gao and Peter Steenkiste, "An adaptive protocol for efficient support of range queries in dht-based systems," in *IEEE ICNP 2004*, October 2004.